A TECHNIQUE FOR TRANSLATING CLAUSAL SPECIFICATIONS OF NUMERICAL METHODS INTO EFFICIENT PROGRAMS

W. F. CLOCKSIN

▷ We describe a technique for translating numerical algorithms specified as clauses into dataflow graphs. The graphs have the property that common subexpressions are computed only once. The purpose of this technique is to convert high-complexity (exponential) solutions derived from elegant clausal specifications into very efficient computations having low (linear or log) complexity. The translation is not a program transformation, but a compilation of a term deduced from a goal clause. The effect of this translation is demonstrated for an assortment of numerical algorithms, including the fast Fourier transform, solution of matrix equations, and series approximation. ◁

1. INTRODUCTION

The application of logic programming to numerical methods has been neither widely explored nor appreciated. Apart from a paper [3] which illustrates some ideas in program transformation using numerical integration as an example, a paper [2] that investigates the parallel execution of communicating processes using a Dirichlet problem as a benchmark program, and a paper [4] showing that a Dirichlet problem may be solved in a declarative manner using constraints, there appears to have been little previous work focusing on logic-programming formulations of numerical methods. We shall attempt to redress this imbalance a little by presenting declarative formulations of some classical numerical methods, and describing an implemented compiler that translates term deduced from these formulations into very efficient dataflow graphs. Such graphs can be used as the program for a hypothetical dataflow computer.

Our method is a kind of partial computation. First, the problem is formulated using clauses, written here in a subset of PROLOG. Such a formulation, though

Address correspondence to Mr. W. F. Clocksin, Computer Laboratory, University of Cambridge, New Museum Site, Pembroke Street, Cambridge CB2 3QG, England.

Received 5 May 1987; accepted 15 October 1987.

THE JOURNAL OF LOGIC PROGRAMMING

52 Vanderbilt Ave., New York, NY 10017 0743-1066/88/$3.50

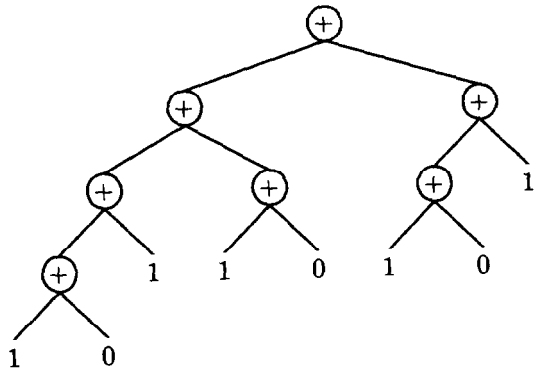


FIGURE 1. Pattern of subgoals for the goal `fib(5,N)`. Multiple identical subgoals are the standard characteristic of the naive definition of `fib`.

elegant, is unlikely to be efficient as the sole means by which solutions can be deduced. However, we use the clausal program to construct a term which, when compiled and executed later, yields the solution. The compiler itself is simply an algorithm for rewriting the term as a directed acyclic graph.

A simple example of the method, for illustrative purposes only, is the well-known naive double-recursive formulation of the Fibonacci sequence, where the goal `fib(`x`,` n`)` succeeds when the xth Fibonacci number can be computed by evaluating term n. This can be written, without loss of generality, in PROLOG as follows:

```
fib(0,0).
fib(1,1).
fib(X,F1+F2) :-
        N1 is N-1, N2 is N-2,
        fib(N1,F1), fib(N2,F2).
```

Satisfying the goal `fib(5,X)` will instantiate `X` to the term depicted in tree form in Figure 1. The tree demonstrates the characteristic shortcoming of actually using such naive formulations in practice: the presence of many identical subgoals. On a sequential computer, repeated execution of an identical subgoal can be considered a waste of time; on a parallel computer, there are additional consequences such as congestion (wasteful occupancy of distribution bandwidth).

Our approach is to compile the resulting tree into the form of a directed acyclic dataflow graph in which common subgoals are merged into one subgoal. Presenting the goal `fib(5,X)` to our compiler causes `X` to be instantiated to the term shown in Figure 2. This is the type of subgoal structure one would expect from the standard iterative Fibonacci algorithm.

The main purpose of the compiler is to merge common subgoals. This is a standard technique in optimizing compilers that eliminate common subexpressions, and the compiler (listed in Appendix A) is a variant of an algorithm for doing this by generating DAGs from intermediate code triples [1]. However, what makes this work interesting is that naive clausal formulations of important numerical methods can also result in efficient programs by applying our technique. We shall now demonstrate several applications in increasing order of sophistication.

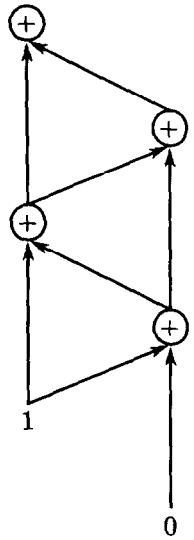


FIGURE 2. Pattern of subgoals resulting from applying the compiler described in this paper to the goal `fib(5,N)`. This pattern is characteristic of iterative algorithms with explicit use of state variables.

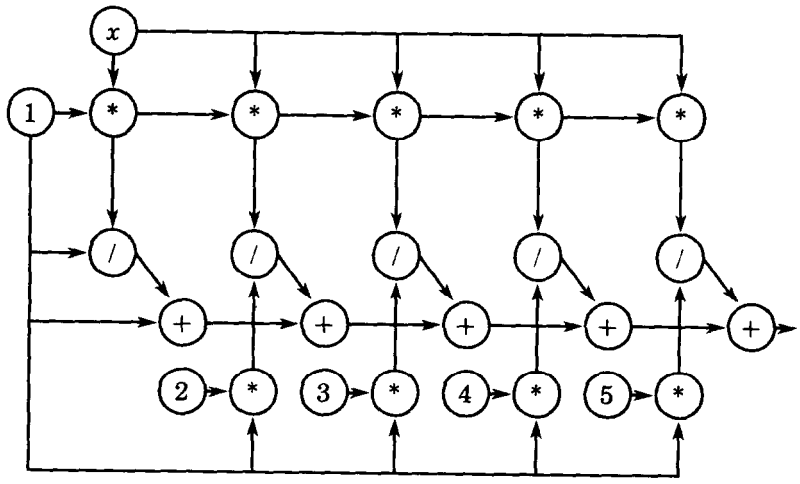
2. APPROXIMATION OF SERIES

The next example, like `fib` above, is a trivial example for illustrative purposes only. The exponential function may be approximated by calculating several terms of the well-known series expansion

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots . \tag{1}$$

The most naive clausal formulation of Equation (1) calculates each factorial and

FIGURE 3. Pattern of subgoals resulting from applying the compiler to the goal `exp(x,5,Y)`. Although the definition of `exp` was a naive but elegant one, this pattern is characteristic of linear algorithms. Some multiplication by constants seen here are easily removed by postprocessing.



power from scratch for each term, and the following clauses do this. The goal `exp(x, n, t)` instantiates t to the tree which, when evaluated, yields the approximation of e^x to n terms:

```
exp(_,0,1).
exp(X,N,XP/TF+S) :-
        power(X,N,XP), fact(N,TF), N1 is N-1, exp(X,N1,S).
power(X,0,1).
power(X,N,X*P) :- N1 is N-1, power(X,N1,P).
fact(1,1).
fact(N,N*F) :- N1 is N-1, fact(N1,F).
```

It should be clear that any tree resulting from such a naive program is grossly unsuitable for efficient computation. However, presenting the goal `exp(x,5,X)` to our compiler causes `X` to be instantiated to the tree shown in Figure 3. This is the type of subgoal structure one would expect from a more efficient algorithm; the number of operations is linear in the number of terms. Some redundant computations can be observed in Figure 3, in particular the bottom row of multiplications. Such nodes are easily removed by postprocessing the dataflow graph using standard techniques. In this case, a postprocessing pass to fold constants suffices to produce the most efficient computation for approximating the exponential function according to Equation (1).

The primary disadvantage of our technique is readily apparent in this example. The compiler generates a term from a given goal. Thus, for the above example it is necessary to know—at compile time—the number of terms of Equation (1). For more serious applications this shortcoming is not always relevant. For many matrix and transform problems of the kind given next, the size of the problem (rank of the matrix, number of transform dimensions, order of the polynomial, etc.) is known at compile time, and it is worthwhile to apply our technique if the resulting program is to be executed more than once.

3. SOLUTION OF MATRIX EQUATIONS

The problem is to solve for the vector $\mathbf{x}$ in the matrix equation $\mathbf{Ux} = \mathbf{a}$, where $\mathbf{U}$ is an upper triangular matrix, and $\mathbf{x}$ is of length n. Each element x_i of $\mathbf{x}$ is given by the following equation:

$$x_i = \frac{1}{U_{i,i}}\left(a_i - \sum_{j=i+1}^{n} x_j U_{i,j}\right). \tag{2}$$

A naive formulation of this equation calculates each x_i independently, ignoring the possibility of performing backsubstitutions. Predicate `solve` is defined such that goal `solve(i, n, s)` succeeds when s is the tree which, when evaluated, names the solution of the n-vector x_i:

```
solve(I,N,(1/u(I,I))*(a(I)-S)) :-
        J is I+1,
        sum(J,N,I,S,N).
```

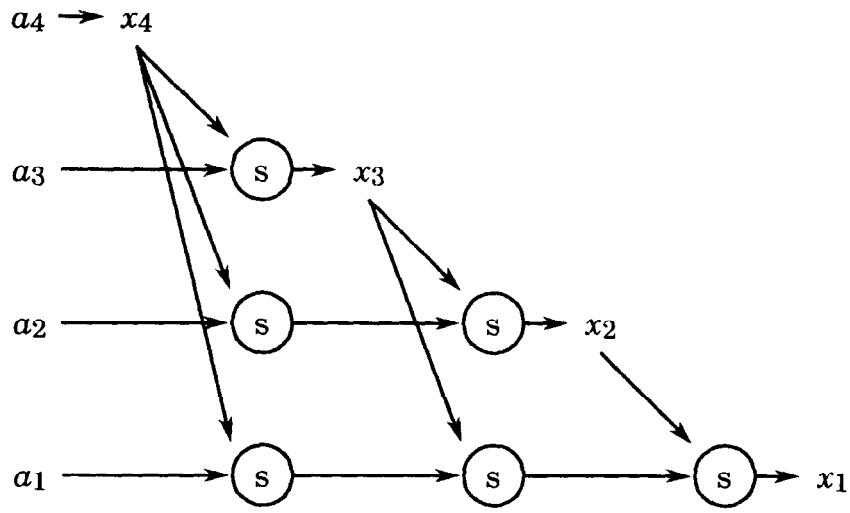


FIGURE 4. Schematic pattern of subgoals resulting from applying the compiler to the four goals `solve(i,X_i,4)` for $i = 1$ to 4. Some detail has been suppressed in the interests of clarity. Although backsubstitution was not specified in the original definition of `solve`, it is clear that the resulting pattern of subgoals uses backsubstitution.

```
sum(A,B,_,0,_) :- A > B.

sum(A,B,I,X*u(I,A)+S,N) :-
     solve(A,X,N), C is A+1, sum(C,B,I,S,N).
```

With this formulation it is necessary to call `solve` with each value of i. This is needlessly extravagant, as it is obvious from Equation (2) that solving for x_i will cause solutions for all x_j, $i < j$, to be calculated. The more usual method of solution involves backsubstitution, which is much more efficient. If the goals `solve(`i, n, s_i`)` for $i = 1$ to n are given to our compiler, the resulting dataflow graph describes a computation which performs the backsubstitutions. The dataflow graph is shown schematically in Figure 4. In this diagram, detail (such as computing the reciprocals of the diagonal elements of **U**) has been suppressed so that the backsubstitution pattern can be seen. Figure 4 shows the pattern of how the a_i are used, how the x_i are calculated, and where the backsubstitutions (the nodes labeled 's') occur. The actual dataflow graph contains some redundant computations (additions by 0) which are easily removed by standard constant-folding techniques.

This example shows how backsubstitutions can be inferred by merging subgoals common to several independent `solve` goals. The only information required at compile time is the order of the matrix. This is often a reasonable requirement in practice.

For Equation (2), because it turns out that all the necessary information is actually present for a solution of x_1 alone, it is not strictly necessary to call `solve` with values of $i > 1$. However, the next example shows a case where all the necessary information is not contained in one goal; common subgoals are distributed among the trees for the independent goals.

4. THE FAST FOURIER TRANSFORM

An n-point discrete Fourier transform (DFT) algorithm can be specified in the following way. Let $p(x)$ be a polynomial in x of degree $n-1$, where n is 2^m for some m:

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}.$$

We shall notate the polynomial $p(x)$ of degree $n-1$ in the following form, where the $i_0, i_1, \ldots i_{n-1}$ are called indices:

$$p_{[i_0, i_1, \ldots, i_{n-1}]}(x) = a_{i_0} + a_{i_1}x + a_{i_2}x^2 + \cdots + a_{i_{n-1}}x^{n-1}.$$

For example,

$$p_{[1,3,5,7]}(x) = a_1 + a_3x + a_5x^2 + a_7x^3.$$

This notation has a practical benefit that will become obvious later when describing the clausal formulation.

Letting ω^i denote the ith power of the nth root of unity, we wish to compute all the $p(\omega^0), p(\omega^1), \ldots, p(\omega^{n-1})$. The computation of a $p(\omega^k)$ proceeds by recursively decomposing a given polynomial into the sum of two polynomials according to the Danielson-Lanczos lemma:

$$p_{[i_0, i_1, \ldots i_{n-1}]}(\omega^k) = p_{[i_0, i_2, \ldots i_{n-2}]}(\omega^{2k}) + \omega^k p_{[i_1, i_3, \ldots i_{n-1}]}(\omega^{2k}).$$

Note that this amounts to recursively rewriting a polynomial having n indices into two polynomials each having the $n/2$ alternating indices of the original polynomial. The recursion terminates when only one index is encountered, in which case we rewrite this to an expression consisting of the indexed coefficient:

$$p_{[i]}(\omega^k) = a_i.$$

4.1. Example: 8-point DFT

Let $p(x)$ be a polynomial of degree 7 in x:

$$p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7.$$

$p(x)$ can be rewritten in the form

$$p(x) = p_{[0,2,4,6]}(x^2) + xp_{[1,3,6,7]}(x^2),$$

where

$$p_{[0,2,4,6]}(x) = a_0 + a_2x + a_4x^2 + a_6x^3,$$

$$p_{[1,3,5,7]}(x) = a_1 + a_3x + a_5x^2 + a_7x^3.$$

Letting ω^i denote the ith power of the 8th root of unity, we wish to compute the following: $p(\omega^0)$, $p(\omega^1)$, $p(\omega^2)$, $p(\omega^3)$, $p(\omega^4)$, $p(\omega^5)$, $p(\omega^6)$, and $p(\omega^7)$.

Now rewrite each polynomial in ω^i according to the above scheme (for simplicity we will not use the identities $\omega^4 = -\omega^0$, $\omega^5 = -\omega^1$, $\omega^6 = -\omega^2$, and $\omega^7 = -\omega^3$):

$$p(\omega^0) = p_{[0,2,4,6]}(\omega^0) + \omega^0 p_{[1,3,5,7]}(\omega^0),$$

$$p(\omega^1) = p_{[0,2,4,6]}(\omega^2) + \omega^1 p_{[1,3,5,7]}(\omega^2),$$

$$p(\omega^2) = p_{[0,2,4,6]}(\omega^4) + \omega^2 p_{[1,3,5,7]}(\omega^4),$$

$$p(\omega^3) = p_{[0,2,4,6]}(\omega^6) + \omega^3 p_{[1,3,5,7]}(\omega^6),$$

$$p(\omega^4) = p_{[0,2,4,6]}(\omega^0) + \omega^4 p_{[1,3,5,7]}(\omega^0),$$

$$p(\omega^5) = p_{[0,2,4,6]}(\omega^2) + \omega^5 p_{[1,3,5,7]}(\omega^2),$$

$$p(\omega^6) = p_{[0,2,4,6]}(\omega^4) + \omega^6 p_{[1,3,5,7]}(\omega^4),$$

$$p(\omega^7) = p_{[0,2,4,6]}(\omega^6) + \omega^7 p_{[1,3,5,7]}(\omega^6).$$

Proceeding with the next recursion,

$$p_{[0,2,4,6]}(\omega^0) = p_{[0,4]}(\omega^0) + \omega^0 p_{[2,6]}(\omega^0),$$

$$p_{[0,2,4,6]}(\omega^2) = p_{[0,4]}(\omega^4) + \omega^2 p_{[2,6]}(\omega^4),$$

$$p_{[0,2,4,6]}(\omega^4) = p_{[0,4]}(\omega^0) + \omega^4 p_{[2,6]}(\omega^0),$$

$$p_{[0,2,4,6]}(\omega^6) = p_{[0,4]}(\omega^4) + \omega^6 p_{[2,6]}(\omega^4).$$

Next,

$$p_{[1,3,5,7]}(\omega^0) = p_{[1,5]}(\omega^0) + \omega^0 p_{[3,7]}(\omega^0),$$

$$p_{[1,3,5,7]}(\omega^2) = p_{[1,5]}(\omega^4) + \omega^2 p_{[3,7]}(\omega^4),$$

$$p_{[1,3,5,7]}(\omega^4) = p_{[1,5]}(\omega^0) + \omega^4 p_{[3,7]}(\omega^0),$$

$$p_{[1,3,5,7]}(\omega^6) = p_{[1,5]}(\omega^4) + \omega^6 p_{[3,7]}(\omega^4).$$

Finally,

$$p_{[0,4]}(\omega^0) = a_0 + \omega^0 a_4,$$

$$p_{[0,4]}(\omega^4) = a_0 + \omega^4 a_4;$$

$$p_{[2,6]}(\omega^0) = a_2 + \omega^0 a_6,$$

$$p_{[2,6]}(\omega^4) = a_2 + \omega^4 a_6;$$

$$p_{[1,5]}(\omega^0) = a_1 + \omega^0 a_5,$$

$$p_{[1,5]}(\omega^4) = a_1 + \omega^4 a_5;$$

$$p_{[3,7]}(\omega^0) = a_3 + \omega^0 a_7,$$

$$p_{[3,7]}(\omega^4) = a_3 + \omega^4 a_7.$$

4.2. *Naive Implementation of the DFT*

We write a root of unity raised to a power k as the compound term `w^k` using an infix operator "^". We write a polynomial $p_{[i_0, i_1, \ldots i_{n-1}]}(w^k)$ as the compound term `p([`$i_1, i_2, \ldots, i_n$`],w^k)`, so for example the polynomial $p_{[0,2,4,6]}(\omega^6)$ is written as `p([0,2,4,6],w^6)`.

As each recursive decomposition requires the even and odd indices, we first define the predicate `alternate`, such that the goal `alternate(L,L1,L2)` succeeds when `L1` is the set of odd indices in `L`, and `L2` is the set of even indices in `L`. The procedure consists of the following two clauses:

```
alternate([],[],[]).
alternate([A,B|T],[A|T1],[B|T2]) :- alternate(T,T1,T2).
```

By inspection of their heads, these two clauses are mutually exclusive. Finally, we define the predicate `eval`. The goal `eval(P,X,N)` succeeds when `X` is the expression which specifies the evaluation of polynomial `P` at a complex root of unity. To compute an n-point DFT, an `eval` goal must be satisfied at each of the `N` powers of the `N` roots of unity. The `eval` procedure consists of the following two clauses:

```
eval(p([I],V),a(I),_).
eval(p(L,V^P),A1+V^P*A2,N) :-
        alternate(L,L1,L2),
        P1 is (P*2) mod N,
        eval(p(L1,V^P1),A1,N),
        eval(p(L2,V^P1),A2,N).
```

The first clause specifies the base case for the recursion. The second clause is the recursive case, which composes the sum-and-product term (in its second argument), finds the alternating indices, multiplies the power, and recurs on the two decomposed polynomials. These two clauses are mutually exclusive (the `alternate` goal fails if its first argument is a one-element list).

As an example, the following goal evaluates the input polynomial at an 8th root of unity ω^6, which is one of eight goals required for an 8-point DFT:

```
eval(p([0,1,2,3,4,5,6,7],w^6),X,8).
X=a(0)+w^0*a(4)+w^4*(a(2)+w^0*a(6))
   +w^6*(a(1)+w^0*a(5)+w^4*(a(3)+w^0*a(7)))
```

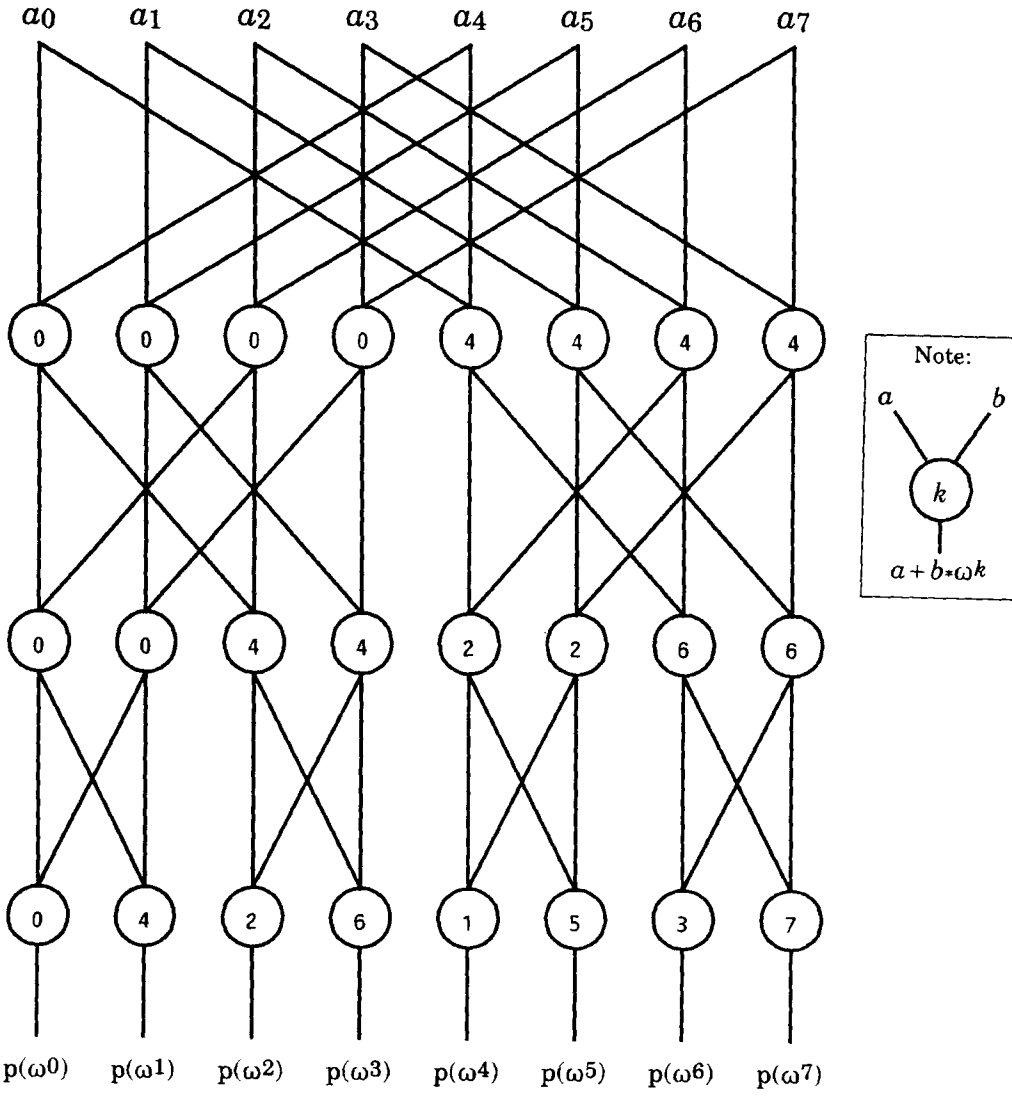
Note that the identities $\omega^4 = -\omega^0$, $\omega^5 = -\omega^1$, etc. are not used, but it is a trivial matter to substitute these if it is ever considered necessary to reduce the number of different complex constants.

4.3. *From DFT to FFT*

Although there are no shared subgoals in the above value of `X`, there are a number of identical subgoals shared among the trees for evaluation of the other seven 8th roots of unity required for the full transform. The key to the *fast* Fourier transform

(FFT) is the saving of work by computing identical subgoals once only. Conventional formulations of the FFT algorithm (for example in [5]) are greatly complicated by the need to deal with multiple copies of the same subgoal. Typically, the common subgoals are computed first, and assigned to certain elements of an array from where they can be accessed later. This standard technique is entirely foreign to the world of logic programming. Because the eight `eval` goals required for an

FIGURE 5. Schematic pattern of subgoals resulting from applying the compiler to the eight goals `eval(p([0,1,2,3,4,5,6,7],w^i),X,8)` for $i = 0$ to 7. Observe the characteristic butterfly pattern of subgoals normally found in the more complicated standard methods for the FFT. In this example, all arithmetic operations are complex ones, and the ω^i are easily calculated complex constants.



8-point FFT are independent, and because the clausal formulation abstracts away from notions of data storage, the *raison d'être* of the FFT is not met by the `eval` predicate alone.

However, the purpose of our compiler is to merge common subgoals wherever they may be found in the input, and this is enough to obtain the effect of the FFT without sacrificing the elegance of the original specification. By giving the trees obtained from the eight eval goals to our compiler, the dataflow diagram of Figure 5 is obtained (an example run for a four-point FFT is shown in Appendix B). In the schematic form shown, each node adds one input to the product of the other input and a root of unity (a complex constant). The pattern of arcs given by our compiler is precisely the "butterfly" pattern characteristic of the conventional (complicated) algorithm for the fast Fourier transform. The only information required by the compiler is the number of points of the transform. This is a reasonable assumption in practice: in some signal-processing applications, the number of points of the transform is hardwired into the apparatus!

5. CONCLUSIONS

The method we have presented here requires a clausal formulation of the numerical method together with a goal clause specifying the desired computation. Clauses (written here in PROLOG syntax) are used simply as an elegant formulation which abstracts away from data-storage models. The goal is satisfied to obtain a pattern of subgoals, which is then compiled to produce a more efficient pattern. Certain parameters need to be known at compile time, and this makes our method less general than the ideal of program transformation (which continues to elude researchers). However, the parameters are related to the size of the problem, and this is reasonable in practice [the order of the matrix in Equation (2), and the number of points of the Fourier transform]. This suggests that our technique is more appropriate for certain design automation problems, for example, in which an elegant but inefficient specification is evaluated to produce a design which is to be executed many times.

The clausal specifications of the above examples can be given a purely declarative status by rewriting them in CLP($\mathbb{R}$). This amounts to changing each "`is`" goal to CLP's " = " constraint.

In the examples shown here, the more efficient patterns have been of lower complexity than the original specification. We are not in a position to make any general claims about the effect the method has on the reduction of computational complexity, and this might inspire further research. The most remarkable result is the automatic derivation of the FFT butterfly pattern.

APPENDIX A. THE GOAL COMPILER

The compiler is a simplified (and much more elegant) version of the algorithm for constructing directed acyclic graphs given in [1]. In the program below, a node of the graph is represented by the binary term `n(`n_1`, t)`, where n_1 is an integer unique node identifier which names the result computed by the node, and t is either a constant (possibly complex, for example ω^6, or possibly subscripted, for example

a(1)) or a compound term of the form $op(n_2, n_3)$, where op describes the computation performed by the node, and n_2 and n_3 are the identifiers of the nodes that compute the arguments for node n_1.

The compiler consists of three procedures. The procedure **gen** succeeds if for goal **gen**(e, l_a, l_b, v), e is the input expression, the output list of dataflow nodes is represented by the difference list composed from l_a and l_b, and v is an accumulator variable used to generate unique node names. The procedure is written with one clause for each operator to be encountered in the input expression. The procedure is intended to be deterministic, and so the cuts are harmless:

```
gen(X+Y,L0,L3,A) :- !,
        gen(X,L0,L1,A1),
        gen(Y,L1,L2,A2),
        node(n(A,A1+A2),L2,L3).
gen(X*Y,L0,L3,A) :- !,
        gen(X,L0,L1,A1),
        gen(Y,L1,L2,A2),
        node(n(A,A1*A2),L2,L3).
gen(X-Y,L0,L3,A) :- !,
        gen(X,L0,L1,A1),
        gen(Y,L1,L2,A2),
        node(n(A,A1-A2),L2,L3).
gen(X/Y,L0,L3,A) :- !,
        gen(X,L0,L1,A1),
        gen(Y,L1,L2,A2),
        node(n(A,A1/A2),L2,L3).
gen((X;Y),L0,L2,_) :- !,
        gen(X,L0,L1,_),
        gen(Y,L1,L2,_).
gen(X,L0,L1,A) :- node(n(A,X),L0,L1).
```

The procedure **node** succeeds if for goal **node**(n, l_a, l_b), n is a computation node encountered in the input expression, and the current list of dataflow nodes is represented by the difference list composed from l_a and l_b. The procedure is intended to be deterministic, and so the cuts are harmless:

```
node(n(1,N),[],[n(1,N)]) :- !.
node(N,L,L) :- find(N,L), !.
node(n(A1,N1),[n(A,N)|T],[n(A1,N1),n(A,N)|T]) :-
     A1 is A+1.
```

The procedure `find` is a simple deterministic check for membership of a list:

```
find(X,[X|_]) :- !.
find(X,[_|T]) :- find(X,T).
```

APPENDIX B. EXAMPLE RUN: 4-POINT FFT

With `gen` as defined in Appendix A, and `eval` as defined in Section 4.2, and assuming the declaration of the infix operator "^", we have

```
:-eval(p([0,1,2,3],w^0),X0,4),
  eval(p([0,1,2,3],w^1),X1,4),
  eval(p([0,1,2,3],w^2),X2,4),
  eval(p([0,1,2,3],w^3),X3,4),
  gen((X0;X1;X2;X3),[],L,_).
L = [n(24,14+23),n(23,22*17),n(22,w^3),n(21,5+20),
     n(20,12*9),n(19,14+18),n(18,15*17),n(17,6+16),
     n(16,12*7),n(15,w^1),n(14,1+13),n(13,12*3),
     n(12,w^2),n(11,5+10),n(10,2*9),n(9,6+8),
     n(8,2*7),n(7,a(3)),n(6,a(1)),n(5,1+4),
     n(4,2*3),n(3,a(2)),n(2,w^0),n(1,a(0))]
```

I thank Martin Richards, Arthur Norman, John Hughes, and Tom Körner for comments and advice.

REFERENCES

1. Aho, A. V. and Ullman, J. D., *Principles of Compiler Design*, Addison-Wesley, 1977.
2. Butler, R., Lusk, E., McCune, W., and Overbeek, R., Parallel Logic Programming for Numerical Applications, in: E. Shapiro (ed.), *Third International Conference on Logic Programming* Lecture Notes in Computer Science 225, Springer, 1986, pp. 375–388.
3. Clark, K. L., W. M. McKeeman, and S. Sickel, Logic program specification of numerical integration, in: K. L. Clark and S.-Å. Tärnlund, (eds.), *Logic Programming*, Academic, 1982, pp. 123–139.
4. Heintze, N., Michaylov, S., and Stuckey, P., CLP(**R**) and Some Electrical Engineering Problems, in: *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, 1987.
5. Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T., *Numerical Recipes*, Cambridge U.P., 1986.